Energy, Mines and Resources Canada

Énergie, Mines et Ressources Canada

CANMET

Canada Centre for Mineral and Energy Technology

Centre canadien de la technologie des minéraux et de l'énergie

Coal Research Laboratories

Laboratoires de recherche sur le charbon

Canada

PIT OPTIMIZATION IN THE PROLOG COMPUTER LANGUAGE

N.J. Stuart

Division Report CRL 89-84 (TR)
October, 1989

This document is an unedited research report prepared primarily for discussion and internal reporting purposes. It does not represent final expression of the opinion of the Canada Centre for Mineral and Energy Technology (CANMET)

24 pp

MICROMEDIA

90-02550/

PIT OPTIMIZATION IN THE PROLOG COMPUTER LANGUAGE

by

N.J. Stuart*

ABSTRACT

The Lerchs-Grossmann Algorithm is exceedingly hard to implement in Fortran, possible to implement in languages such as C or Pascal but almost trivial in Prolog. A program is listed and explained. With modification to use recursive structures speed of solution in a compiled Prolog shouldn't be much worse than other languages. Unfortunately memory becomes a problem in the MS-DOS environment.

However, there are various techniques which should greatly simplify large models including the compositing of overburden blocks, removal of blocks which can be easily proven to be in or out of the solution and the compositing of pay blocks.

This demonstrates the power of Prolog to address problems concerned with finite graphs.

*Research Scientist, Canada Centre for Mineral and Energy Technology, Department of Energy Mines and Resources Canada, P.O. Bag 1280, Devon, Alberta, Canada, TOC 1EO.

OPTIMISATION DE L'EXPLOITATION MINIERE EN LANGAGE INFORMATIQUE PROLOG

par

N.J. Stuart*

RÉSUMÉ

L'algorithme Lerchs-Grossmann est excessivement difficile à mettre en application en Fortran, il peut être mis en application dans le langage C ou le Pascal, mais son application en Prolog est simple au point d'être presque banale. Le contenu d'un programme est présenté et des explications sont fournies. Avec une modification en vue d'utiliser des structures récursives la vitesse de résolution en langage Prolog compilé devrait être aussi bonne qu'avec d'autres langages. Malheureusement, avec le MS-DOS, la mémoire devient un problème.

Toutefois, il existe diverses techniques qui devraient simplifier grandement les modèles de grandes dimensions, notamment l'assemblage de blocs de morts-terrains, l'élimination de blocs qui ne peuvent pas être facilement identifiés comme faisant partie ou non de la solution, et l'assemblage de blocs exploitables.

Ces possibilités montrent la puissance du Prolog pour résoudre les problèmes relatifs à des graphiques finis.

---

*Chercheur scientifique, Centre canadien de la technologie des minéraux et de l'énergie, Énergie, Mines et Ressources Canada, C.P. 1280, Devon (Alberta), Canada T0C 1E0

CONTENTS

	Page
ABSTRACT	i
RESUME	iii
INTRODUCTION	1
IMPLEMENTATION PROBLEMS	2
IMPLEMENTING THE LG METHOD IN PROLOG	2
IMPROVING THE PERFORMANCE	7
ANOTHER PHILOSOPHY	8
SIMPLIFICATION AND AUTOMATIC PAYBLOCK AGGLOMERATION	11
NOTE ON OCTATREES	12
PROLOG AND GRAPHS	12
CONCLUSIONS	13
REFERENCES	13
APPENDIX A	A-14
APPENDIX B	B-17

NO. FIGURES

1a	Block Model Showing Values	19
1b	Block Model Showing "G": the slope criterion or dependence removal graph	19
2a	Agglomeration of overburden blocks into sets. Note A&B&C=().	19
2b	Set A contains cone B - set B is empty	19
3a	Plan view of three cone intersection on surface. Arrows represent the dependency graph "G"	20
3b	Graph "G" derived from FIG. 3a.	20

INTRODUCTION

In 1964 a paper was published (1) that covered several aspects of mine economics including a 2 and a 3 dimensional approach to pit optimization. The 2d case is simple to understand and implement, however, it has no real practical use. The three dimensional solution is given in an Appendix and is here after described as the Lerchs-Grossmann (LG) method.

The LG method is rooted in graph theory. It requires as input a set of block values (i.e. the profit that can be made from a block of ore, minus the cost of mining it) and a set of relationships describing which blocks are supported by which other blocks, such that, if block 'a' is mined block 'b' must be mined first. The set of such constraints is hereafter referred to as 'the graph of G' or just 'G'; member arcs of G will be written g(lower block,upper block). We generally consider G to be the graph of directly effected blocks such that if we have elements g(a,b) and g(b,c) we do not include g(a,c) although clearly the dependency exists. A graph derived from G that included these cases would be termed a 'transitive closure' and would be a very large set.

It is generally assumed that one is dealing with a block model, i.e. a set of cuboid units laid out like the unit cells in an orthorhombic crystal (Figure 1A p.20). The LG method does not assume that and any implementation that assumes this is a restricted case.

It is further generally assumed that there is a simple 'slope constraint' that enables G to be determined (Figure 1B p. 20). If blocks in our cubic models are named by 3-tuples of integers in the form b(Easting,Northing,Depth) then the slope constraint generates members of G from b(Easting,Northing, Depth) to the 9 blocks b(E,N,Depth-1) where E takes the value Easting-1,Easting and Easting+1 and N takes the values Northing-1, Northing and Northing+1. In other words to remove a block one must first remove the block directly above and its 8 immediate neighbors. This is attractive to the theoretician since G need not be stored explicitly but can be easily calculated as needed. Such simple slope constraints, however, severely limit the usefulness of the method to one slope angle throughout the pit (determined by the shape of the block). Further a single pay block will generate a pit that is a square pyramid rather than a cone. These constraints are often quoted as justification for using heuristic (non optimal) methods.

For the LG method one adds a dummy root block (called x0 here) and creates a new graph containing all of the nodes of G (but different arcs). This structure is always a tree (and thus will be called 'tree' here) though it changes in the course of the solution. One starting point (adopted here) is that the tree is initially a set of arcs connecting the root node x0 to each block in the model. The solution consists of a series of iterations that manipulate the tree until the ultimate pit is found.

IMPLEMENTATION PROBLEMS

It is extremely inconvenient to represent and manipulate tree structures in FORTRAN. Languages such as C or Pascal contain features specifically designed to permit it. A Prolog program, however, is itself a tree and Prolog systems contain an in-built tree traversal mechanism. The result is that it is orders of magnitude easier to program graph theory problems in Prolog than any other language. The code tends to be tight and self explanatory. It is, however, not the most efficient language from the point of view of computer resources (CPU time memory etc.). The question arises as to which is more valuable given that the saved programmer time in the Prolog case can be used to increase the sophistication of the algorithm.

IMPLEMENTING THE LG METHOD IN PROLOG

A Prolog language consists of a set of rules. The simplest type of rule is the fact. To represent the value of blocks one might choose facts of the form:

```
node(nodename,value).
e.g. node (x0,-1)
```

or to represent a member of G:

```
g(nodename,nodename2).
e.g. g(x0,x1).
```

or the node names themselves might be composite to indicate their position:

g(b(easting,northing,depth),b(easting2,northing2,depth2).
e.g. g(b(21,45,12),b(20,46,11).

In the case of the block model one might prefer to infer G using the following rules:

```
g(b(E,N,D),b(E2,N2,D2)):-
neigh(E,Ev),
neigh(N,Nv),
D2=D-1.

neigh(X,X1):- X=X1+1.
neigh(X,X).
neigh(X,X1):- X=X1-1.
```

Given the coordinates of the lower block, this rule will generate all 9 blocks above. In essence it thus contains two nested loops.

The representation of structures and data in the program will not be covered in detail here. The demonstration program is admittedly primitive in this regard paying no attention to performance. It was written in Turbo Prolog V1 (a product of Borland International) which does not permit "recursive structures". Turbo Prolog V2 does permit them as do most systems that adhere more closely to the Edinburgh standard. Steps required to enhance the performance of the system will be given later.

Writing the program largely consisted of translating definitions in the original paper into Prolog rules. The data used consisted of three types of fact:

Value data in facts called "node", in the form:

```
node (x0,-1).
```

G is represented by a set of facts such as:

```
arc(x2,x1).
```

and the tree in the form

```
tree(x0,x1).
```

The first definitions were that a pair of blocks is "plus" if it exists in the same order in tree as in G. And minus if the order is reversed. This gives us:-

```
plus (X,Y):-tree(X,Y),
            arc(X,Y).

minus(X,Y,):-tree(X,Y),
             arc(X,Y).
```

The only complicated rule to write was the definition of 'support'. The value supported by a block is the sum of its value and those of the blocks in the branch that stems from it. The Prolog rule may be read as follows:

Node Y supports mass (value) M where:-(first find the value of node Y and remember it as M), find all the nodes Z such that tree(Y,Z) and store them in a list L, use the rule "Vallist" to determine the sum of the masses supported by the members of the list. The rule "Vallist" can be thought of as being doubly recursive; it determines the value of each node by using the rule 'support' (which was the rule that called itself). Secondly "Vallist" calls itself in what is a conventional manner for Prolog rules to handle lists. In Prolog this translates to:-

```
support(Y,M):-

        node(Y,Mnode),
        findall(Z,tree(Y,Z),L),
        vallist(L,0,Val),
        M=Mnode+Val.
        vallist([ ],V,V).
        vallist([H|T],V,Vf):-

              support(H,M),
              Vn=V+M,
```

```
!,vallist(T,Vn,Vf).
```

Hardly a lot of code for a traversal of a multi-branch tree (binary trees which can have only 2 branches per node are much simpler).

The next definition is of a strong pair in the tree. Tree (X,Y) is strong if it is plus and it supports a positive value OR if it is minus and it supports a value of zero or less. In Prolog:

```
strong(X,Y):-
        plus(X,Y),
        support(Y,V),
        V>0.

strong(X,Y):-
        minus(X,Y),
        support(Y,V),
        V<=0
```

likewise a vertex (block) is strong if it is at the branchward end of a strong tree connection:

```
strongv(V):-
        tree(X,V),
        strong(X,V).
```

The LG algorithm really consists of 4 rules, the first definition covered in the text is normalization. To normalize a tree one finds strong tree pairs in which the first node is not the root of the graph x0 and splits the graph up at this point by eliminating (retracting) the strong tree fact and creating (asserting) a new tree fact linking the root node x0 to the branch that would otherwise be disconnected. This is continued until no such strong tree facts can be found.

```
normalize:-
        strong(X,Y),
        not(root(X)),
```

```
        retract(tree(X,Y)),
        root(R),
        asserta(tree(R,Y)),
        asserta(arc(R,Y)),
        !,normalize.
normalize.
```

The LG algorithm itself is as follows. Repeat the following on a normalized tree:-

1) find a pair of blocks (X,Y) such that X is strong (X,Y) is an arc in G and Y is not strong.
2) Remove the basal tree fact from below X (i.e. tree (x0,B) where B supports the branch that contains X. Add a tree fact Y,X).
3) Normalize the tree.

Part 1 is performed by the rule lgtest and part 2 by flip.

```
lg:-
        lgtest(X,Y),
        flip(X,Y),
        normalize,
        !,lg.

lg:-write("solution found -sum of strong nodes"),nl.

lgtest(X,Y):-
        strongv(X),
        arc(X,Y),
        not(strongv(Y)).

flip(X,Y):-
        root(X),
        retract(tree(X,Y)),!.
```

```
flip(X,Y):-
        tree(Z,X),
        retract(tree(X,Y)),
        assertz(tree(Y,X)),
        !,flip(Z,X).

flip(X,Y):-
        tree(Z,X),
        assertz(tree(Y,X)),
        !,flip(Z,X).
```

This is the complete algorithm in Prolog. The solution is the set of blocks supported by strong tree components. One author stated that it was not "possible or desirable" to implement the LG method on a digital computer.

IMPROVING THE PERFORMANCE

Let us take performance to mean the ability to solve a given set of block values and G graph with the minimum computer resources. The performance aims for such a program include speed of solution and reasonable use of memory. In particular one must be concerned with the combinatorial effects and how the resource requirements grow with model size.

Clearly where one is dealing with a block model one would first replace the arc facts with a rule, thus G would be inferred rather than specified. This would save space as well as time as the lgtest searches the entire arc data base looking for ones commencing with the node X. There can be almost 9 times as many arcs in G as there are blocks.

The tree structure can be changed so that each node contains the value which is supports. This would remove needless multiple uses of the time consuming support rule. These values would have to be updated on effected nodes during the normalization and flipping process. Nevertheless the saving would be great.

The representation of the tree structure itself is interesting. There are the same number of tree "facts" as there are blocks. A real world block model would likely be of the order of 100,000 blocks. One would like to use a recursive structure. Prolog has a structure called a list. Each element of a list

can be a list and so on. Unfortunately, if one is using MS-DOS one is restricted for most purposes to 640K of memory. Clearly this consideration will disappear in the near future. For the time being, one must review the possibilities.

1) Many applications will fit into 640K even with Prologs extravagant use of memory.
2) One must consider the use of memory beyond the 640K limit.

There are three options:

i) One effectively declares expanded memory as a RAM disk and uses it as a way to store "facts". There could be various keying systems that would greatly speed the search speed.
ii) One could implement trees in files. Turbo Prolog supports binary trees in files. Clearly it is possible to convert any tree to a binary tree with some wasted memory.
iii) MS-DOS V4 is rumoured to "break the 640K barrier". It is not clear what this means or whether systems such as Turbo Prolog would immediately gain access to the extra memory in a useful form.

In summary, there are a vareity of steps that can be taken so that a Prolog LG program could be made to give acceptable performance.

ANOTHER PHILOSOPHY

It is my intention to show how one could move away from the block model paradigm.

Let us assume that one has a very large block model. Suppose there was:

1) A method to group payblocks into units so that if any one member of the group was mined, the others would also be mined. This might be a purely inspectional process or, as will be shown later, it could be computerized.
2) For any such group (which may not be touching each other) there is a method for determining the overburden cone. For the time being, the term "overburden cone" is defined as a set of blocks that

must first be excavated. Obviously this can be done by any realistic geotechnical criterion.

3) There is some means for performing boolean algebra on the sets of blocks.

Any set of blocks that is not bisected by the excavation cone of a pay zone can be considered as a single block. Thus, if in the case of three pay zones there are a maximum of seven possible composite blocks (Figure 2 P20).

Blocks that are in all three cones. Blocks that are in the combined cones of a&b(but not c), b&c(but not a),c&a(but not b), and blocks that are in a,b and c only. Further we can write down G (Figure 2 P21). If we call these blocks a&b&c,a&b etc. then

g(a,a&b).
g(a,a&c).
g(b,b&c).
g(b,a&b).
g(c,a&c).
g(c,b&c).
g(b&c,a&b&c).
g(a&b,a&b&c).
g(a&c,a&b&c).

These are always the sets for a three cone system, some of the sets may be empty, however, for example, if a and b do not intersect then there is no set a&b (or rather it is empty). If, for example, cone b is completely contained by cone a then cone b is empty -one has block a and block a&b.

Prolog is strong at combinatorial problems such as this. For demonstration, we have replaced each ore body with a point beneath it such that the cone from that point would just contain the ore body and no more. In this simple case, we allow a selectable slope angle. We define orebodies with the fact "payblock" which contains an integer to name the block and a three dimensional coordinate. The rule "intersect(AA,B)" finds cones that intersect at zero depth if the payblock number for B is greater than that of A. The complete set of possible blocks is found if for each known block (initially the set of payblocks) a block is added for each intersecting pay block. Thus for pay blocks 1,2,3,4., if 2 intersects with block 1 since "intersection only looks at

higher numbered blocks), one adds blocks 2&3, 2&4, 2&5, 2&6 and 2&7. Now considering 2&3 we consider that there may be a block 2&3&4 if 2 intersec ts 4 and 3 intersects 4. This raises the slight possibility that an empty block is added since it is possible to have three cones such that each pair intersects but there is no triple intersection. However, we will have to address the issue of empty blocks later anyway.

The relationship for the G graph can also be expressed as a recursive relationship so it again can be represented implicitly.

This block simplification is attractive and simple but has one shortcoming. Crowding of payblocks causes a combinatorial explosion. Obviously it is the number of block interactions that counts. A quite small number of pay blocks can cause problems if each intersect with each other. On the other hand one can deal with very large sets of payblocks if each has only a moderate number of intersections.

Before dealing with techniques for dealing with this problem, let us discuss what has been achieved in this section. The block grouping technique has enabled any slope angle to be used. Here we have represented pits as cones of constant slope angle and vertical axes. Any other slope criterion could be used provided that one can simply calculate which excavations would intersect.

Secondly, provided that one can remove empty blocks there will always be an improvement in the number of blocks to be handled, and quite clearly this will usually be a massive improvement.

Thirdly, the groupings have a physical meaning. It is of interest to know which volumes of overburden are open up multiple ore bodies rather than just one. In particular, such composite blocks are of interest in performing discounted cash flow calculations.

Finally, though I have been referring to a block model throughout, this approach could clearly be extended to any geometric representations of geology provided that it allowed the intersecting volumes of excavations to be calculated.

If a block is found to be empty then it can be removed. For this there must be an accommodation in the G graph. Each arc entering the removed block must be replaced by a set of arcs going to each of the destinations of arcs leaving the removed block.

SIMPLIFICATION AND AUTOMATIC PAYBLOCK AGGLOMERATION

Here we address the issue of preventing combinatorial explosions in the block compositing process described above. There are two main ways to simplify the model. One is to remove blocks from consideration, the other is to combine blocks.

One seeks to use the LG method to obtain the provable optimum pit. Heuristic methods may (1) fail to find profitable blocks, (2) mine non-profitable blocks or (3) do both. Types 2 and 3 may be useful. The choice of heuristic is a function of the type of deposit. Take type (1) methods. Suppose one can find a set of blocks that can be shown to be part of the optimal pit, for example, by extracting payblocks that by themselves pay for the removal of their overburden without help from other payblocks. The removal of such a block will relieve the overburden on other blocks and thus make them provably profitable by this technique. Such a technique can be proven never to remove blocks not in the ultimate pit. We can remove all such blocks from our model and put them in a "solution file". By "mining out" the core of the deposit we eliminate crowded areas of of cone interaction perhaps even splitting the problem into discrete sub problems. Type (2) heuristics can be used to identify blocks that can never be profitable and thus can just be retracted from the problem. For example, a block whose value is less than its unshared overburden cannot be in the optimal pit and can thus be removed. This in turn increases the size of other blocks unshared overburden and may lead to their removal. Deep rogue blocks bring lots of extra overburden blocks into the model and increase the intersection combinatorial problem. There are thus 2 types of removal methods.

If block a would be profitable, if block b is removed and block b is profitable if block a is removed then we may treat them as one (composite) block. More complicated algorithms for finding composite blocks are possible but suffer from combinatorial problems. Curiously one can write the profitability of a block as a set of (logical) horn clauses (the form of a Prolog program):

profitable(a) if b,c,d. (are mined first).

or

profitable(a) if c,d,e. (are mined first).

The process of producing composite blocks becomes aggravated at depth where one needs to aggregate large numbers of blocks. Block compositing can be run iteratively with the removal techniques above. If a composite block can be shown to be either in or out of the solution it can be removed. The removal of blocks, in turn, may facilitate the identification of candidate composite blocks.

NOTE ON OCTATREES

Various authors have noted that in certain geological situations memory can be conserved by using structures known as octatrees. An Octatree is a recursive structure. An element in an octatree (usually a cube) represents a homogenous geological region and it merely stores values that represent the geology of that region. If the region is inhomogenous it is divided into 8 smaller cubes which are in turn treated as elements. It would clearly be simple to create and manipulate octatrees in Prolog. It would form a natural partner to a LG algorithm.

PROLOG AND GRAPHS

The multipurpose nature of a Prolog program suggests that one can compile a set of clauses that could tackle most commonly posed graph theory problems. Indeed much of the code may already exist in one form or another, however, there is little dialogue between the logic programming and operations research fraternities. In most cases the graphs to be processed are much smaller than a mines block model thus the performance issue does not apply.

Why then is Prolog so adept at graph problems? Possible reasons include:-

1) The flow pattern built into Prolog is easily manipulated to search trees, etc.
2) The ease of use of recursion and recursive structures is extremely helpful.
3) Prolog, as a logic programming language, is good at problems that are logically rather than numerically complex.

CONCLUSIONS

The ease, brevity and elegence of the Prolog implementation of the Lerchs Grossman algorithm is surprising. Further, because it is expressed in terms of block values and a removal dependence graph as input, it is truly general and need not be thought of as pertaining exclusively to cuboid block models. The performance of LG implementations with very large block sets can be improved by prior manipulation of the input data by removing and compositing blocks. The effect of these methods could, in the right situations be dramatic. However, the performances of all aspects discussed here are rather involved from a combinatorial point of view, and it is highly likely that two very similar input problems could take vastly different resources to solve.

REFERENCES

1. Lerchs, H., Grossman, I.F., "Optimum Design of Open Pit Mines". Bulletin CIMM, January, 1965, Vol. 68, P. 47-54.

APPENDIX "A"
Annotated listing of Main LG Program

```
/* domains and predicates sections of
program serve to define and declare
structures to the turbo prolog system */
domains
slist=symbol*
database
arc(symbol,symbol)
tree(symbol,symbol)

predicates
node(symbol,integer)
plus(symbol,symbol)
minus(symbol,symbol)
edge(symbol,symbol)
root(symbol)
support(symbol,integer)
countlist(slist,integer,integer)
vallist(slist,integer,integer)
strong(symbol,symbol)
strongv(symbol)
weak(symbol,symbol)
normalize
backwrite(symbol)
lgtest(symbol,symbol)
flip(symbol,symbol)
lg(integer)

/*------ Program section------- */
clauses
/* master rule for lg process */
lg(N):-
        lgtest(X,Y),
        flip(X,Y),
        normalize,
        N1=N+1,
        write("iteration no",N1),nl,
        !,lg(N1).
lg(_):-write("The solution is found"),nl.


/* test to find if further iterations
   are necessary */
lgtest(X,Y):-
        strongv(X),
        arc(X,Y),
        not(strongv(Y)).

/* flip performs step 2 of
lerchs grossman. If X is a strong vertex
Y isn't (such a pair can be found with lgtest)
flip will add tree(Y,X) into the database, reverse
the tree entries for the branch supporting Y
and snip off the tree(x0,xm) entry. */

flip(X,Y):-
        root(X),
        retract(tree(X,Y)),!.
flip(X,Y):-
        tree(Z,X),
        retract(tree(X,Y)),
        assertz(tree(Y,X)),
        !,flip(Z,X).
flip(X,Y):-
        tree(Z,X),
        assertz(tree(Y,X)),
        !,flip(Z,X).

/* backwrite is not part of the basic program
   but can be used for examining trees */

backwrite(V):-
        root(V).
backwrite(V):-
        tree(X,V),
        write(X,'-',V),nl,
        !,backwrite(X).

/* normalize adjusts the tree database to produce
normalized trees. The tree must be normalozed at the
start of the problem. The LG rule also uses
normalize with each iteration */

normalize:-
        strong(X,Y),
        not(root(X)),
        retract(tree(X,Y)),
        root(R),
        asserta(tree(R,Y)),
        asserta(arc(R,Y)),
        !,normalize.

normalize.
/* rule defines strong arc */
strong(X,Y):-
        plus(X,Y),
        support(Y,V),
        V>0.
strong(X,Y):-
        minus(X,Y),
        support(Y,V),
        V<=0.

/* rule defines strong vertex */
strongv(V):-
        tree(X,V),
        strong(X,V).
strongv(V):-
        tree(X,V),
        strongv(X).

/* defines weak arc, this is not used by the LG routine
   but is included for interest */
weak(X,Y):-
        tree(X,Y),
        not(strong(X,Y)).
/* edge defines a non directed tree component*/
edge(X,Y):- arc(X,Y).
edge(X,Y):-arc(Y,X).

/* define plus and minus arcs */
/* the following only work if X Y in tree order */
plus(X,Y):-tree(X,Y),
        arc(X,Y).
minus(X,Y):-tree(X,Y),
        arc(Y,X).
/* support and vallist search a tree to find the value
supported */
support(Y,M):-
        node(Y,Mnode),
        findall(Z,tree(Y,Z),L),
        vallist(L,0,Val),
        M=Mnode+Val.

vallist([],V,V).
vallist([H|T],V,Vf):-
        support(H,M),
        Vn=V+M,
        !,vallist(T,Vn,Vf).
```

```
/*countlist is not needed as part of the lg
program but is handy in examining things*/
countlist([],C,C).
countlist([_|T],I,C):-
        Il=I+1,
        !,countlist(T,Il,C).

/*define dummy root*/
root(x0).
/* database of nodes and their value */
/*----------End of 'program' section-----------*/

/*The following databases of facts can be replaced with
other representaions of the information. They represent the
'data' both input and output since the final
solution is obtained by interogating it for
the set of strong vertices. */

node(x0,-1).
node(x1,-4).
node(x2,1).
node(x3,-4).
node(x4,2).
node(x5,-1).
node(x6,5).
node(x7,-1).
node(x8,3).
node(x9,-2).
/* database representing the g graph
   inluding dummy nodes to the dummy root */
arc(x0,x1).
arc(x0,x2).
arc(x0,x3).
arc(x0,x4).
arc(x0,x5).
arc(x0,x6).
arc(x0,x7).
arc(x0,x8).
arc(x0,x9).
arc(x5,x4).
arc(x3,x4).
arc(x3,x2).
arc(x2,x1).
arc(x5,x6).
arc(x6,x7).
arc(x8,x6).
arc(x8,x9).

/* dynamic database that is to be manipulated to
   obtain final solution. These facts change during the
   program */

tree(x0,x1).
tree(x0,x2).
tree(x0,x3).
tree(x0,x4).
tree(x0,x5).
tree(x0,x6).
tree(x0,x7).
tree(x0,x8).
tree(x0,x9).
```

APPENDIX "B"

Listing of Program to Derive Slope Constraint Vectory

```
/*  Program to derive vector set for implicit
representation of G-graph slope criterion */

database
block(integer,integer,integer)
sblock(integer,integer,integer)
predicates
coneset(integer,integer)
conex(integer,integer,integer)
coney(integer,integer,integer,integer)
tana(real)
add(integer,integer,integer)
incone(integer,integer,integer)
showcone(integer,integer)
writeblock(integer,integer)
addall(integer,integer,integer,integer)
blast
same(integer,integer,integer,integer,integer,integer)

/*----------program section----------*/
/* This program generates slope constraint
   vectors. The incone rule set is modified so
   that incone(X,Y,H) is true if a block of set
   by X and Y units in the x and y directions
   and H blocks above a payblock is considered in
   the overburden cone.
      There are three parts to using the program. The
      rule coneset is invoked first. It generates a
      complete set of blocks that are in the over
      burden to level MAX above the payblock. For
      efficiency this is implemented as a scanning
      routine. This would need to be modified if, for
      example, the cone was elongated in a direction not
      parallel with the x or y axis.
        The rule 'blast' removes all blocks that can be
      described as the sum of two or more other block's
      coordinates.
        'Showcone' shows the resultant cone, one level
      at a time to a required level.*/

clauses
/* showcone uses writeblock to print out the
   derived overburden shape to Max levels
   above the payblock */
showcone(0,_):-
        block(X,Y,Z),
        assertz(sblock(X,Y,Z)),
        fail.
showcone(0,Max):-
        !,showcone(1,Max).

showcone(Level,Max):-
        Level<=Max,
        writeblock(Level,Max),
        L1=Level+1,
        !,showcone(L1,Max).
writeblock(L,M):-
        sblock(X,Y,L),
        Xp=X+9,
        YP=Y+10,
         cursor(Xp,Yp),
        write("X"),
        addall(X,Y,L,M),
        retract(sblock(X,Y,L)),
        !,writeblock(L,M).
writeblock(_,_):-readchar(_).
addall(X,Y,L,M):-
        block(X1,Y1,L1),
        L2=L+L1,
        L2<=M,
        X2=X+X1,
        Y2=Y+Y1,
        not(sblock(X2,Y2,L2)),
        assertz(sblock(X2,Y2,L2)),
        !,addall(X,Y,L,M).
addall(_,_,_,_).


/* incone is the rule defining the slope criterion.
   here it is a simple upright cone with angle
   given by tanh. This rule can be modified for
   any other slope constraint.*/

incone(X,Y,H):-
        Hor=sqrt((X*X)+(Y*Y)),
        tana(T),
        Hor<(T*H).
tana(0.65).

/*coneset uses conex and coney to generate
  all the blocks in the cone up to level
  'Max', H is initially 0. */
coneset(H,Max):-
        H>Max.
coneset(H,Max):-
        conex(0,+1,H),
        conex(0,-1,H),
        H1=H+1,
        !,coneset(H1,Max).
conex(X,Inc,H):-
        incone(X,0,H),
        coney(X,0,H,+1),
        coney(X,0,H,-1),
        X1=X+Inc,
        !,conex(X1,Inc,H).
conex(_,_,_).

coney(X,Y,H,Inc):-
        incone(X,Y,H),
        add(X,Y,H),
        Y1=Y+Inc,
        !,coney(X,Y1,H,Inc).
coney(_,_,_,_).

/*adds a block to database if it not already there*/
add(X,Y,Z):-
        not(block(X,Y,Z)),
        assertz(block(X,Y,Z)),!.
add(_,_,_).
/* eliminates all vectors that can be expressed
   as combinations of others */
blast:-
        block(B,N,M),
        block(J,K,L),
        not(same(B,N,M,J,K,L)),
        I=J-B,
        O=K-N,
        P=L-M,
        incone(I,O,P),
        retract(block(J,K,L)),
        !,write("*"),blast.
blast.
/* same used to show two vectors equal */
same(J,K,L,B,N,M):-
        J=B,K=N,L=M.
```

FIGURES

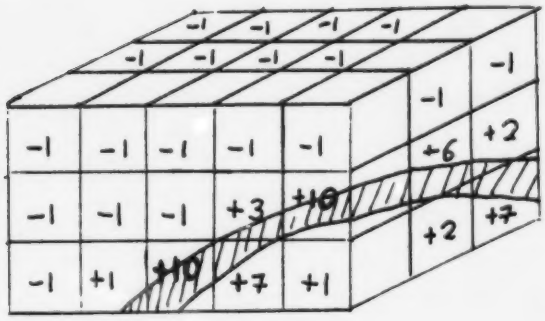

FIGURE 1a: Block Model Showing Values.

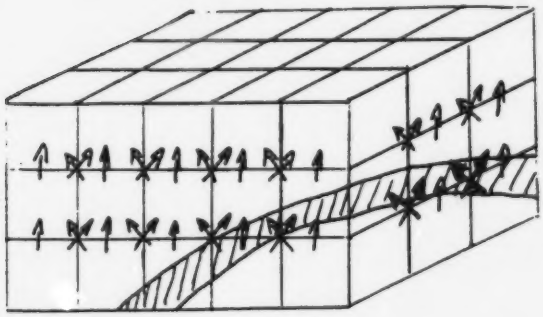

FIGURE 1b: Block Model Showing 'G': the slope criterion or dependence removal graph.

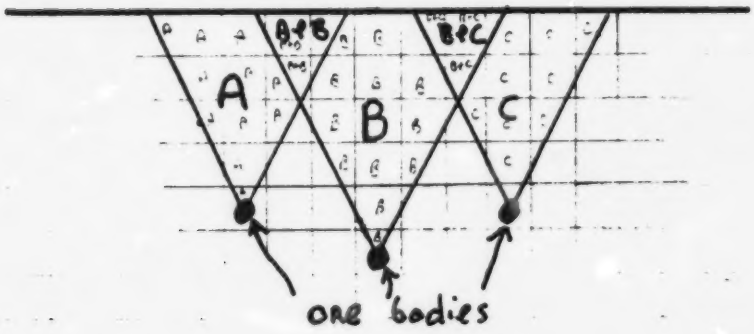

FIGURE 2a: Agglomeration of overburden blocks into sets. Note A&B&C=().

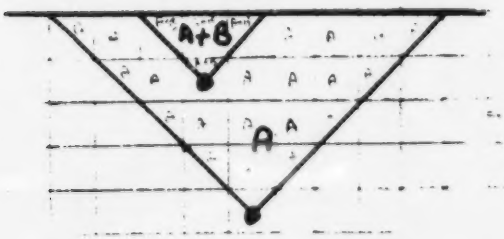

FIGURE 2b: Set A contains cone B -set B is empty.

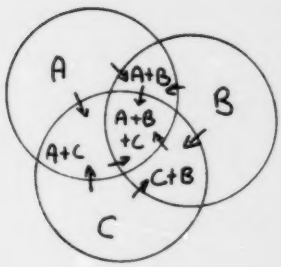

FIGURE 3a: Plan view of three cone intersection on surface. Arrows represent the dependency graph 'G'.

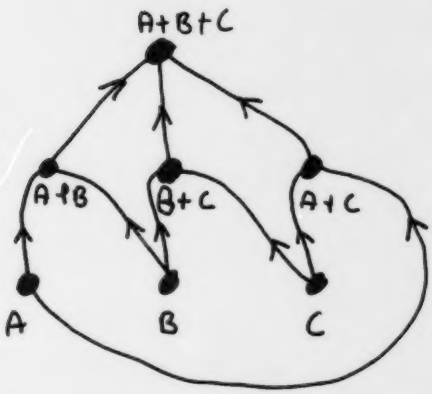

FIGURE 3b: Graph 'G' derived from FIG 3a.